Developing Autonomic Properties for Distributed Pattern-Recognition Systems with ASSL: A Distributed MARF Case Study *

Emil Vassev
Lero - the Irish Software Engineering Research Centre
University of Limerick, Limerick, Ireland
emil@vassev.com

Serguei A. Mokhov
Faculty of Engineering and Computer Science, Concordia University
1455 de Maisonneuve Blvd. W, Montreal, QC, Canada
mokhov@cse.concordia.ca

Abstract

In this paper, we discuss our research towards developing special properties that introduce autonomic behavior in pattern-recognition systems. In our approach we use ASSL (Autonomic System Specification Language) to formally develop such properties for DMARF (Distributed Modular Audio Recognition Framework). These properties enhance DMARF with an autonomic middleware that manages the four stages of the framework's pattern-recognition pipeline. DMARF is a biologically inspired system employing pattern recognition, signal processing, and natural language processing helping us process audio, textual, or imagery data needed by a variety of scientific applications, e.g., biometric applications. In that context, the notion go autonomic DMARF (ADMARF) can be employed by autonomous and robotic systems that theoretically require less-to-none human intervention other than data collection for pattern analysis and observing the results. In this article, we explain the ASSL specification models for the autonomic properties of DMARF.

Keywords: autonomic computing, formal methods, ASSL, DMARF

1 Introduction

Today, we face the challenge of hardware and software complexity that appears to be the biggest threat to the continuous progress in IT. Many initiatives towards complexity reduction in both software and hardware have arisen with the advent of new theories and paradigms. Autonomic computing (AC) [13] promises reduction of the workload needed to maintain complex systems by transforming them into self-managing autonomic systems. The AC paradigm draws inspiration from the human body's *autonomic nervous system* [3]. The idea is that software systems can manage themselves and deal with dynamic requirements, as well as unanticipated threats, automatically, just as the body does, by handling complexity through self-management.

Pattern recognition is a widely used biologically inspired technique in the modern computer science. Algorithms for image and voice recognition have been derived from the human brain, which uses pattern recognition to recognize shapes, images, voices, sounds, etc. In this research, we applied the principles of AC to solve specific problems in distributed pattern-recognition systems, such as availability, security, performance, etc. where the health of a distributed pipeline

*This work was supported in part by an IRCSET postdoctoral fellowship grant (now termed as EMPOWER) at University College Dublin, Ireland, by the Science Foundation Ireland grant 03/CE2/I303_1 to Lero (the Irish Software Engineering Research Centre), and by the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada.

is important. We tackled these issues by introducing self-management into the system behavior. As a proof-of-concept (PoC) case study, we used ASSL [18, 25] to develop the autonomic self-management properties for DMARF [10], which is an intrinsically complex pipelined distributed system composed of multi-level operational layers. The ASSL framework helped us develop the self-managing features first, which we then integrated into DMARF. In this paper, based on our results, we provide an attempt to generalize our experience on any similar-scale distributed pipelined pattern-recognition system.

1.1 Problem Statement and Proposed Solution

Distributed MARF (DMARF) could not be used in autonomous systems of any kind as-is due to lack of provision for such a use by applications that necessitate the self-management requirements. Extending DMARF directly to support the said requirements is a major redesign and development effort to undertake for an open-source project. Moreover, such and extended autonomic DMARF must be validated and tested, since there is no immediate guarantee that the properties the latter has been augmented with are intrinsically correct.

In our approach, we provide the methodology for the initial proof-of-concept. We specify with ASSL a number of autonomic properties for DMARF, such as self-healing [24], self-optimization [23], and self-protection [12]. The implementation of those properties is generated automatically by the ASSL framework in the form of a special wrapper Java code that provides an autonomic layer implementing the DMARF's autonomic properties. In addition, the latter are formally validated with the ASSL's mechanisms for consistency and model checking. Model checking is performed on the generated Java code, where the ASSL relies on the Java PathFinder [2] tool developed by NASA Ames.

1.2 Organization

The rest of this article is organized as follows. In Section 2, we briefly review the field of autonomic computing and describe both ASSL and DMARF frameworks. Section 3 presents details of the ASSL specification models for autonomic properties of DMARF. Finally, Section 4 presents some concluding remarks and future work.

2 Background

The vision and metaphor of AC [13] is to apply the principles of self-regulation and complexity hiding. The AC paradigm emphasizes reduction of the workload needed to maintain complex systems by transforming those into self-managing autonomic systems (AS). The idea is that software systems shall automatically manage themselves just as the human body does. Nowadays, a great deal of research effort is devoted to developing AC tools. Such a tool is the ASSL framework, which helps AC developers with problem specification, system design, system analysis and evaluation, and system implementation.

ASSL was initially developed by Vassev at Concordia University, Montreal, Canada [25] and since then it has been successfully applied to the development of a variety of autonomic systems including distributed ones. For example, ASSL was used to develop autonomic features and generate prototype models for two NASA missions – the ANTS (Autonomous Nano-Technology Swarm) concept mission (where a thousand of picospacecraft work cooperatively to explore the asteroid belt [17]), and the Voyager mission [16]. In both cases, there have been developed autonomic prototypes to simulate the autonomic properties of the space exploration missions and

validate those properties through the simulated experimental results. The targeted autonomic properties were: self-configuring [22], self-healing [19], and self-scheduling [21] for ANTS, and autonomic image processing for Voyager [20]. In general, the development of these properties required a two-level approach, i.e., they were specified at the individual spacecraft level and at the level of the entire system. Because ANTS is intrinsically distributed system composed of many autonomous spacecraft, that case study required individual specification of the autonomic properties of each individual spacecraft member of the ANTS swarm.

2.1 ASSL

The Autonomic System Specification Language (ASSL) [18, 25] approaches the problem of formal specification and code generation of autonomic systems (ASs) within a framework. The core of this framework is a special formal notation and a toolset including tools that allow ASSL specifications be edited and validated. The current validation approach in ASSL is a form of consistency checking (handles syntax and consistency errors) performed against a set of semantic definitions. The latter form a theory that aids in the construction of correct AS specifications. Moreover, from any valid specification, ASSL can generate an operational Java application skeleton.

Overall, ASSL considers autonomic systems (ASs) as composed of autonomic elements (AEs) communicating over interaction protocols. To specify those, ASSL is defined through formalization of tiers. Over these tiers, ASSL provides a multi-tier specification model that is designed to be scalable and exposes a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. The ASSL tiers and their sub-tiers (see Figure 1) are abstractions of different aspects of the AS under consideration. They aid not only to specification of the system at different levels of abstraction, but also to reduction of the complexity, and thus, to improving the overall perception of the system.

There are three major tiers (three major abstraction perspectives), each composed of sub-tiers (see Figure 1):

- *AS tier* – presents a general and global AS perspective, where we define the general autonomic system rules in terms of *service-level objectives (SLO)* and *self-management policies*, *architecture topology* and *global actions*, *events* and *metrics* applied in these rules.
- *AS Interaction Protocol (ASIP) tier* – forms a communication protocol perspective, where we define the means of communication between AEs. An ASIP is composed of *channels*, *communication functions*, and *messages*.
- *AE tier* – forms a unit-level perspective, where we define interacting sets of individual AEs with their own behavior. This tier is composed of AE rules (*SLO* and *self-management policies*), an *AE interaction protocol* (AEIP), *AE friends* (a list of AEs forming a circle of trust), *recovery protocols*, special *behavior models* and *outcomes*, *AE actions*, *AE events*, and *AE metrics*.

The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Figure 1). The AS SLOs are a high-level form of behavioral specification that help developers establish system objectives (e.g., performance). The self-management policies could be any of (but not restricted to) the four so-called self-CHOP policies defined by the AC IBM blueprint: *self-configuring*, *self-healing*, *self-optimizing* and *self-protecting* [4]. These policies are event-driven and trigger the execution of actions driving an AS in critical situations. The metrics constitute a set of parameters and

```
I. Autonomic System (AS)
  * AS Service-level Objectives
  * AS Self-managing Policies
  * AS Architecture
  * AS Actions
  * AS Events
  * AS Metrics
II. AS Interaction Protocol (ASIP)
  * AS Messages
  * AS Communication Channels
  * AS Communication Functions
III. Autonomic Element (AE)
  * AE Service-level Objectives
  * AE Self-managing Policies
  * AE Friends
  * AE Interaction Protocol (AEIP)
    - AE Messages
    - AE Communication Channels
    - AE Communication Functions
    - AE Managed Elements
  * AE Recovery Protocol
  * AE Behavior Models
  * AE Outcomes
  * AE Actions
  * AE Events
  * AE Metrics
```

Figure 1: ASSL Multi-Tier Model

observables controllable by an AS. At the ASIP Tier, the ASSL framework helps developers specify an AS-level interaction protocol as a public communication interface, expressed with special *communication channels*, *communication functions* and *communication messages*. At the AE Tier, the ASSL formal model exposes specification constructs for the specification of the system's AEs.

Conceptually, AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs. These relationships are specified at both ASIP and AEIP tiers. Whereas ASIP specifies an AS-level *interaction protocol* that is public and accessible to all the AEs of an AS and to *external systems* communicating with that very AS, the AEIP tier is normally used to specify a *private communication protocol* used by an AE to communicate only with: 1) trusted AEs, i.e., AEs declared as "AE Friends" (see Figure 1); and 2) special controlled *managed elements*. Therefore, two AEs exchange messages over an AEIP only if they are *friends*, thus revealing the need for special negotiation messages specified at ASIP to discover new friends at runtime.

Note that ASSL targets only the AC features of a system and helps developers clearly distinguish the AC features from the system-service features. This is possible, because with ASSL we model and generate special AC wrappers in the form of ASs that embed the components of non-AC systems. The latter are considered as *managed elements*, controlled by the AS in question. A managed element can be any software or hardware system (or sub-system) providing services. Managed elements are specified per AE (they form an extra layer at the AEIP see Figure 1) where the emphasis is on the control interface. It is important also to mention that the ASSL tiers and sub-tiers are intended to specify different aspects of an AS, but it is not necessary to employ all of them in order to model such a system. For a simple AS we need to specify 1) the AEs providing self-managing behavior intended to control the managed elements associated with an AE; and 2) the communication interface. Here, self-management policies must be specified to provide such self-managing behavior at the level of AS (the AS Tier) and at the level of AE (AE Tier). The self-management behavior of an ASSL-developed AS is specified with the

```
ASSELF_MANAGEMENT {
 SELF_HEALING {
  FLUENT inLosingSpacecraft {
   INITIATED_BY { EVENTS.spaceCraftLost }
   TERMINATED_BY { EVENTS.earthNotified }
}
  MAPPING {
   CONDITIONS { inLosingSpacecraft }
   DO_ACTIONS { ACTIONS.notifyEarth }
  }
 }
} // ASSELF_MANAGEMENT
```

Figure 2: Self-management Policy

self-management policies. These policies are specified with special ASSL constructs termed *fluents* and *mappings* [18, 25]. A fluent is a state where an AS enters with *fluent-activating events* and exits with *fluent-terminating events*. A mapping connects fluents with particular actions to be undertaken. Usually, an ASSL specification is built around self-management policies, which make that specification AC-driven. The policies themselves are driven by events and actions determined deterministically. Figure 2 presents a sample specification of an ASSL self-healing policy.

For more details on the ASSL multi-tier specification model and the ASSL framework toolset, please refer to [18, 25].

2.2 Distributed MARF

DMARF [10] is based on the classical MARF whose pipeline stages were made into distributed nodes. The Modular Audio Recognition Framework (MARF) [5] is an open-source research platform and a collection of pattern recognition, signal processing, and natural language processing (NLP) algorithms written in Java and arranged into a modular and extensible framework facilitating addition of new algorithms for use and experiments by scientists. MARF can run distributively over the network, run stand-alone, or may just act as a library in applications. MARF has a number of algorithms implemented for various pattern recognition and some signal processing tasks. The backbone of MARF consists of pipeline stages that communicate with each other to get the data they need in a chained manner.

In general, MARF's pipeline of algorithm implementations is presented in Figure 3 (where the implemented algorithms are grouped in white boxes, and the stubs or in progress algorithms are grouped in gray). The pipeline consists of the four core stages grouping the similar kinds of algorithms: (1) sample loading, (2) preprocessing, (3) feature extraction, and (4) training/-classification. MARF's distributed extension, DMARF [10] allows the stages of the pipeline to run as distributed nodes as well as a front-end. The basic stages and the front-end were implemented without backup recovery or hot-swappable capabilities at this point; just communication over Java RMI [26], CORBA [14], and XML-RPC WebServices [15]. There is also an undergoing project on the intensional scripting language, MARFL [7] to allow scripting MARF tasks and applications.

There are various applications that test and employ MARF's functionality and serve as examples of how to use MARF. High-volume processing of recorded audio, textual, or imagery data are possible pattern-recognition and biometric applications of DMARF. In this work, most of the emphasis is on audio processing, such as conference recordings with purpose of attribution of said material to identities of speakers. Another emphasis is on processing a bulk of recorded

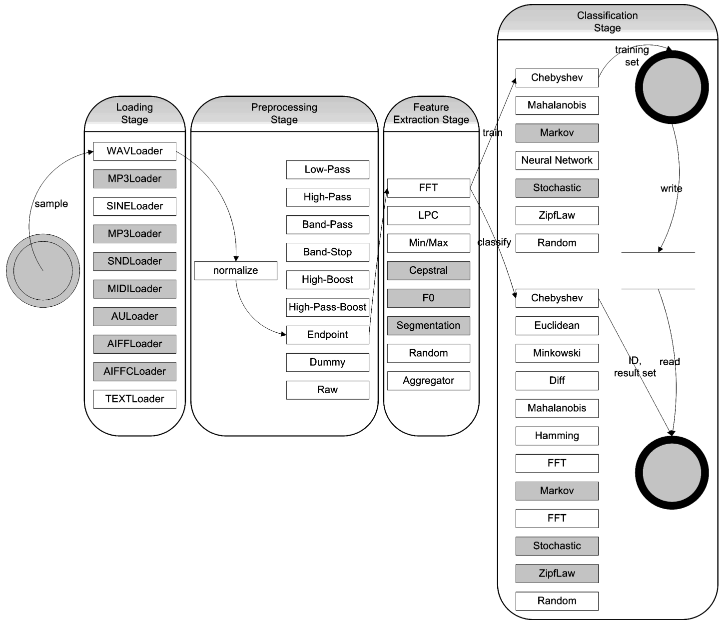

Figure 3: MARF's Pattern Recognition Pipeline

phone conversations in a police department for forensic analysis [8] and subject identification and classification. See the cited works and references therein for more details on MARF and applications.

3 Making Distributed Pipelined Systems Autonomic with ASSL: Autonomic DMARF Case Study

In general, ASSL helps to design and generate special *autonomic wrappers* in the form of AEs that embed one or more system components. The latter are considered as *managed elements* (see Section 2.1) that present one or more *single nodes* of a distributed system. Therefore, for each distributed node, we ideally specify with ASSL a single AE that introduces an autonomic behavior to that node. All the AEs are specified at the *AE Tier* and the global autonomic behavior of the entire system is handled by specifications at the *AS Tier* (see Figure 1). As shown in Section 2.1, we rely on a rich set of constructs, such as *actions*, *events*, and *metrics* to specify special self-management policies driving the nodes of a distributed system in situations requiring autonomic behavior. Moreover, with ASSL, we specify special interaction protocols (ASIP and AEIP) that help those nodes exchange messages and synchronize on common autonomic behavior. In this section we demonstrate how ASSL may be applied to an inherently distributed system such as DMARF. The novelty in our approach is safeguarding the distributed pipeline, which is not possible with plain distributed systems. Therefore, with this case study we not only demonstrate the applicability of ASSL to distributed systems but also validate that ASSL may successfully be applied to *pipelined distributed systems*.

DMARF's capture as an AS primarily covers the autonomic behavior of the distributed pattern-recognition pipeline. We examine properties that apply to DMARF and specify in

detail the self-CHOP aspects of it. If we look a the DMARF pipeline as a whole, we see that there should be at least one instance of every stage somewhere on the network. There are four main core pipeline stages and an application-specific stage that initiates pipeline processing. If one of the core stages goes offline, the pipeline stalls and to recover it has the following options: 1) use of a replacement node; 2) recovery of the failed node; or 3) rerouting the pipeline through a different node with the same service functionality as the failed one.

In order to make DMARF autonomic, we need to add *automicity* (autonomic computing behavior) to the DMARF behavior. We add a special *autonomic manager* (AM) to each DMARF stage. This makes the latter AEs, those composing an autonomic DMARF (ADMARF) capable of self-management.

3.1 Self-Healing

A DMARF-based system should be able to recover itself through replication to keep at least one route of the pipeline available. There are two types of replication: 1) the replication of a service, which essentially means that we increase the number of nodes per core stage (e.g. two different hosts provide preprocessing services as active replication, so if one goes down, the pipeline is still not stalled; if both are up they can contribute to load balancing, which is a part of the self-optimization autonomic property); and 2) replication within the node itself. If all nodes of a core stages go down, the stage preceding it is responsible to start up a temporary one on the host of the preceding stage, set it up to repair the pipeline. This is the hard replication needed to withstand stall faults, where it is more vulnerable and not fault-tolerant. In the second case, denoting passive replication of the same node (or even different nodes) losing a primary or a replica is not as serious as in the first case because such a loss does not produce a pipeline stall and it is easier to self-heal after a passive replica loss. Restart and recovery of the failed node without replicas is another possibility for self-healing for DMARF. Technically, it may be tried prior or after the replica kicks in.

In the course of this project, we used ASSL to specify the self-healing behavior of ADMARF by addressing specific cases related to *node replacement* (service replica) and *node recovery*, shown in Figure 4.

The following sub-sections describe the ASSL specification of the self-healing algorithm revealed here. We specified this algorithm as an ASSL self-healing policy spread on both system (AS tier) and autonomic element (AE tier) levels where *events*, *actions*, *metrics*, and special *managed element interface functions* are used to incorporate the self-healing behavior in ADMARF (see Appendix A). Note that due to space limitations Appendix A presents a partial ASSL specification where only one AE (DMARF stage) is specified. The full specification specifies all the four DMARF stages.

3.1.1 AS Tier Specification for Self-Healing

At the AS tier we specify the global ADMARF self-healing behavior. To specify the latter, we use an ASSL `SELF_HEALING` self-management policy (see Figure 5). Here we specified a single *fluent* mapped to an *action* via a *mapping*.

Thus, the `inLowPerformance` fluent is initiated by a `lowPerformanceDetected` event and terminated by one of the events such as `performanceNormalized` or `performanceNormFailed`. Here the `inLowPerformance` event is activated when special AS-level `performance` service-level objectives (SLO) degrade (see Figure 6). Note that in ASSL, SLO are evaluated as *Booleans* based on their *satisfaction* and thus, they can be evaluated as *degraded* or *normal* [25]. Therefore, in our specification model, the `lowPerformanceDetected` event is activated anytime

```
ADMARF monitors its run-time performance and in case of performance degradation
  notifies the problematic DMARF stages to start self-healing;
Every notified DMARF stage (note that this is an AE) analyzes the problem locally to
  determine its nature: a node is down or a node is not healthy (does not perform well);
if A node is down then
    // The following node-replacement algorithm is followed by the AM of
       the stage:
    AM strives to find a replica note of the failed one;
    if replica found then
        next redirect computation to it;
    end
    if replica not found then
        report the problem. Note that the algorithm could be extended with a few more
        steps where the AM contacts the AM of the previous stage to organize pipeline
        reparation;
    end
end
if A node does not perform well then
    // The following node-recovery algorithm is followed:
    AM starts the recovery protocol for the problematic node;
    if recovery successful then
        do nothing;
    end
    if recovery unsuccessful then
        AM strives to find a replica node of the failed one;
    end
    if replica found then
        next redirect computation to it;
    end
    if replica not found then
        report the problem;
    end
end
```

Figure 4: DMARF Self-Healing Algorithm

when the ADMARF's performance goes down. Alternatively, the `performanceNormalized` event activates when the same performance goes up.

As specified, the AS-level `performance` SLO are a global task whose realization is distributed among the AEs (DMARF stages). Thus, the AS-level performance degrades when the performance of any of the DMARF stages goes down (see the `FOREACH` loop in Figure 6), thus triggering the `SELF_HEALING` policy. In addition, the `performanceNormFailed` event activates if an a special event (`selfHealingFailed`) occurs in the system. This event is specified at the AE tier (see Section 3.1.2) and reports that the local AE-level self-healing has failed. Although not presented in this specification, the `performanceNormFailed` event should be activated by any of the `performanceNormFailed` events specified for each AE (a DMARF stage).

Moreover, once the `inLowPerformance` fluent gets initiated, the corresponding `startSelfHealing`

```
ASSELF_MANAGEMENT {
  SELF_HEALING {
     // a performance problem has been detected
    FLUENT inLowPerformance {
      INITIATED_BY { EVENTS.lowPerformanceDetected }
      TERMINATED_BY { EVENTS.performanceNormalized,
                      EVENTS.performanceNormFailed }
    }
    MAPPING {
      CONDITIONS { inLowPerformance }
      DO_ACTIONS { ACTIONS.startSelfHealing }
    }
  }
}  // ASSELF_MANAGEMENT
```

Figure 5: AS Tier SELF_HEALING Policy

```
ASSLO {
  SLO performance {
    FOREACH member in AES {
      member.AESLO.performance
    }
  }
}
....
EVENTS {  // these events are used in the fluents specification
  EVENT lowPerformanceDetected {
    ACTIVATION { DEGRADED { ASSLO.performance } } }
  EVENT performanceNormalized {
    ACTIVATION { NORMALIZED { ASSLO.performance } } }
  EVENT performanceNormFailed {
    ACTIVATION {
      OCCURRED { AES.STAGE_AE.EVENTS.selfHealingFailed } } }
}  // EVENTS
```

Figure 6: AS Tier SLO and Events

action is executed (see Figure 6). This action simply triggers an AE-level event if the performance of that AE is degraded. The AE-level event prompts the `SELF_HEALING` policy at the AE level (see Section 3.1.2).

3.1.2 AE Tier Specification for Self-Healing

At this tier we specify the self-healing policy for each AE in the ADMARF AS. Recall that the ADMARF 's AEs are the DMARF stages enriched with a special autonomic manager each.

Appendix A presents the self-healing specification of one AE called `STAGE_AE`. Note that the latter can be considered as a generic AE and the specifications of the four AEs (one per DMARF stage) can be derived from this one. Similar to the AS-level specification (see Section 3.1.1), here we specify a (but AE-level) `SELF_HEALING` policy with a set of *fluents* initiated and terminated by *events* and *actions* mapped to those *fluents* (see Appendix A). Thus we specified three distinct fluents: `inActiveSelfHealing`, `inFailedNodesDetec-ted`, and `inProblematicNodesDetected`, each mapped to an AE-level action. The first fluent gets initiated when a `mustDoSelfHealing` event occurs in the system. That event is triggered by the AS-level `startSelfHea-ling` action in the case when the `performance` SLO of the AE get degraded (see Appendix A).

Here the `performance` SLO of the AE are specified as a Boolean expression over two ASSL

```
AESLO {
  SLO performance {
    METRICS.numberOfFailedNodes
    AND
    METRICS.numberOfProblematicNodes
  }
}
```

Figure 7: AE Tier SLO

```
VALUE { 0 }
THRESHOLD_CLASS { Integer [0] }  // valid only when holds 0
```

Figure 8: AE Tier Metric Threshold Class

metrics, such as the `numberOfFailedNodes` metric and the equivalent `numberOfProblematicNodes` (see Figure 7). Whereas the former measures the number of *failed notes* in the DMARF stage, the latter measures the number of *problematic nodes* in that stage.

Both metrics are specified as `RESOURCE` metrics, i.e., observing a managed resource controlled by the AE [25]. Note that the managed resource is the DMARF stage itself. Thus, as those metrics are specified (see Appendix A) they get updated by the DMARF stage via special *interface functions* embedded in the specification of a `STAGE_ME` managed element (see Appendix A). In addition, both metrics are set to accept only a zero value (see Figure 8), thus set in the so-called metric `THRESHOLD_CLASS` [25]. The latter determines rules for *valid* and *invalid* metric values. Since in ASSL metrics are evaluated as Booleans (valid or invalid) based on the value they are currently holding, the performance SLO (see Figure 7) gets degraded if one of the two defined metrics, the `numberOf-FailedNodes` metric or the `numberOfProblematicNodes` metric become *invalid*, i.e., if the DMARF stage reports that there is one or more *failed* or *problematic* nodes.

The `inActiveSelfHealing` fluent prompts the `ana-lyzeProblem` action execution (see Appendix A). The latter uses the `STAGE_ME` managed element's interface functions to determine the nature of the problem – is it a node that failed or it is a node that does not perform well. Based on this, the action triggers a `mustSwitch-ToNodeReplica` event or a `mustFixNode` event respectively. Each one of those events initiates a fluent in the AE `SELF_HEALING` policy to handle the performance problem. The `inFailedNodesDetected` fluent handles the case when a node has failed and its replica must be started and the `inProblematicNodesDetected` fluent handles the case when a node must be recovered. Here the first fluent prompts the execution of the `start-ReplicaNode` action and the second prompts the execution of the `fixProblematicNode` action. Internally, both actions call interface functions of the `STAGE_ME` managed element. Note that those functions trigger erroneous events if they do not succeed (see Figure 9). Those events terminate fluents of the AE `SELF_HEALING` policy (see Appendix A).

It is important to mention that the `inFailedNodes-Detected` fluent gets initiated when the `mustSwitchTo-NodeReplica` event occurs in the system. The latter is triggered by the `analyzeProblem` action.

Moreover, the same event activates, according to its specification (see Figure 10), if a `nodeCannotBeFixed` event occurs in the system, which is due to the inability of the `recoverNode` interface function to recover the problematic node (see Figure 9). Therefore, if a node cannot be recovered the `inFailedNodesDetected` fluent will be initiated in an attempt to start the replica of that node. Note that this conforms to the self-healing algorithm presented in Section 3.

```
// runs the replica of a failed node
INTERFACE_FUNCTION runNodeReplica {
  PARAMETERS { DMARFNode node }
  ONERR_TRIGGERS { EVENTS.nodeReplicaFailed }
}
// recovers a problematic node
INTERFACE_FUNCTION recoverNode {
  PARAMETERS { DMARFNode node }
  ONERR_TRIGGERS { EVENTS.nodeCannotBeFixed }
}
```

Figure 9: AE Tier STAGE_ME Functions

```
EVENT mustSwitchToNodeReplica {
  ACTIVATION {
    OCCURRED { EVENTS.nodeCannotBeFixed }
  }
}
```

Figure 10: Event mustSwitchToNodeReplica

3.2 ASSL Self-Protection Model for DMARF

For scientific and research computing on a local network in a controlled lab environment, runs of DMARF do not need to be protected against malicious alteration or denial of service. However, as soon as the researchers across universities need to cooperate (or police departments to share the audio data recognition or computing), various needs about security and protection arise about data and computation results integrity, confidentiality, and the fact they come from a legitimate source. Therefore, *self-protection* of a DMARF-based system is less important in the localized scientific environments, but is a lot more important in global environments ran over the Internet, potentially through links crossing country borders. This is even more true if the data being worked on by a DMARF installation are of a sensitive nature such as recordings of the phone conversations of potential terrorist suspects. Thus, we point out the general requirements for this autonomic property of DMARF:

- For the self-protection aspect, the DMARF-based systems should adhere to the specification where each node proves its identity to other nodes participating in the pipeline as well as passive replicas. This will insure the data origin authentication (the data are coming from a legitimate source) and will protect against spoofing of the data with distorted voice recordings or incorrect processed data at the later stages of the pipeline. Thus, we ensure the trustworthiness of the distributed data being processed [6]. This can be achieved by proxy certificates issued to the nodes during the deployment and management phase. Each node digitally signs the outgoing data, with the signature the recipient node can verify through the certificate of the sender signed by trusted authority. This is in a way similar to how DNSSec [1] operates for the DNS names and servers by attaching a public key to the host and IP pair signed by the higher-level domain or authority. The similar trust mechanism is also important when DMARF is used for scientific research installation that say crosses the boundaries of several Universities' network perimeters over the Internet while performing scientific computation – the bottom line the data coming from the pipeline stages should be trustworthy, i.e. correct.
- The same proxy certificates can also help with the data privacy along public channels,

especially when identities of real people are involved, such as speakers, that cross the Internet. The system should protect itself from any falsification attempt by detecting it, halting the corresponding computation, and logging and reporting the incident in a trustworthy manner.

- Run-time communication protocol selection is a self-protection option that ensures availability in case the default communication mechanism becomes unavailable (e.g. the default port of `rmiregistry` becomes blocked by a firewall) the participating nodes switch to XML-RPC over HTTP. The protection of self against Distributed Denial of Service (DDoS) is a difficult problem, which extends onto protecting not only self to the best available means but also the self's peers by avoiding flooding them by the self's own output of a compromised node. While the DDoS attacks are very difficult to mitigate if a node is under attack, each node can protect self and others by limiting the amount of outgoing traffic it, itself, produces when a compromise is suspected or too much traffic flood is detected.

For self-protection, DMARF-based systems should adhere to the specification where each node proves its identity to other nodes participating in the pipeline as well as passive replicas. This will insure the data origin authentication (the data are coming from a legitimate source) and will protect against spoofing of the data with distorted voice recordings or incorrect processed data at the later stages of the pipeline. Thus, we ensure the trustworthiness of the distributed data being processed [6]. This can be achieved by proxy certificates issued to the nodes during the deployment and management phase. Each node digitally signs the outgoing data, with the signature the recipient node can verify through the certificate of the sender signed by trusted authority. This is in a way similar to how DNSSec [1] operates for the DNS names and servers by attaching a public key to the host and IP pair signed by the higher-level domain or authority. The same proxy certificates can also help with the data privacy along public channels, especially when identities of real people are involved, such as speakers, that cross the Internet. The system should protect itself from any falsification attempt by detecting it, halting the corresponding computation, and logging and reporting the incident in a trustworthy manner.

To provide self-protecting capabilities, DMARF has to incorporate special autonomic computing behavior. To achieve that, similar to our related work on the self-healing and self-optimization models for DMARF [23, 24], we add a special autonomic manager (AM) to each DMARF stage. This converts the latter into AEs that compose the autonomic DMARF (ADMARF) capable of self-management. Self-protecting is one of the self-management properties that must be addressed by ADMARF . Here we use ASSL to specify the self-protecting behavior of ADMARF where incoming messages must be secure in order to be able to process them. Thus, if a message (public or private) is about to be received in the AS, the following self-protection algorithm is followed by the AM of the stage (AE level) for private messages or by the global AM (AS level) for public messages:

- A message hook mechanism detects when a message (public or private) is about to be received.
- AM strives to identify the sender of that message by checking the embedded digital signature:
  - If the message does not carry a digital signature then it is considered insecure.
  - If the message carries a digital signature then its digital signature is checked:

  - ∗ If the digital signature is recognized then the message is considered secure.
  - ∗ If the digital signature is not recognized then the message is considered insecure.

- If the message is secure no restrictions are imposed over the IO operations and the message can be processed further.
- If the message is insecure the message is simply discarded by blocking any IO operations over that message.

The following sections describe the DMARF specification of the self-protecting algorithm revealed here. We specified this algorithm as an ASSL self-protecting policy spread on both system (AS tier) and autonomic element (AE tier) levels where *events*, *actions*, *metrics*, and special *managed element interface functions* are used to incorporate the self-protecting behavior in ADMARF (see Appendix B). In addition, two interaction protocols – a public (ASIP tier) and a private (AEIP tier), are specified to provide a secure communication system used by both DMARF nodes and external entities to communicate. Note that due to space limitations Appendix B presents a partial ASSL specification where only one AE (DMARF stage) is specified. The full specification specifies all the four DMARF stages.

3.2.1 IP Tiers Specification

Recall that ASSL specifies AEs as entities communicating via special *interaction protocols* (see Section 2.1). Note that all the communication activities (sending and receiving messages), all the communication channels, and all the communication entities (ASSL messages) must be specified in order to allow both internal and external entities to communicate. Hence, no entity can either send or receive a message that is not an ASSL-specified message or use alternative mechanism of communication. Thus, for the needs of the self-protecting mechanism, we specified two communication protocols – at the ASIP tier and at the AEIP tier (this is nested in the AE specification structure) (see Section 2.1). Please refer to Appendix B for a complete specification of both protocols.

At the ASIP tier, we specified a single public message (called `publicMessage`), a single sequential bidirectional public communication channel (called `public-Link`), and two public communication functions; specifically `receivePublicMessages` and `sendPublicMessa-ges`. They are specified to receive and send public messages over the public channel. Here any message sent or received must be an instance of the ASSL-specified `publicMessage`. The latter has an embedded `Proxy-Certificate` parameter specified to carry the digital signature of the message sender (see Appendix B). This parameter plays a key role in the self-protecting behavior of ADMARF . Every message sender must complete this parameter with its *proxy certificate* before sending the message or the latter will be discarded by the system.

Moreover, the mentioned communication functions (`receivePublicMessages` and `sendPublicMessages`) are the only ones specified in the entire AS able to process instances of the `publicMessage` ASSL message. Thus, to process such a message both functions are equipped with a conditional clause to check if the message is secure (see Figure 11).

Figure 11 shows the `receivePublicMessages` communication function. As is depicted, in order to send a public message the `thereIsInsecurePublicMessage` metric (see Appendix B) must be *valid*. The latter is updated by the self-protecting policy and is considered *invalid* (its operational evaluation returns `FALSE` [25]) if the public message to be received is *insecure*.

Note that the specification of the AEIP tier is identical to that of the ASIP tier (see Appendix B), but deals with private messages [25], i.e., external entities cannot send or receive such messages.

```
 //receive public messages if the message is secure
FUNCTION receivePublicMessages {
  DOES {
    IF ( AS.METRICS.thereIsInsecurePublicMessage ) THEN
      MESSAGES.publicMessage << CHANNELS.publicLink
    END
  }
}
```

Figure 11: ASSL Specification of receivePublicMessages

```
SELF_PROTECTING {
   // a new incoming message has been detected
  FLUENT inSecurityCheck {
    INITIATED_BY { EVENTS.publicMessageIsComing }
    TERMINATED_BY { EVENTS.publicMessageSecure,
                    EVENTS.publicMessageInsecure }
  }
  MAPPING {
    CONDITIONS { inSecurityCheck }
    DO_ACTIONS { ACTIONS.checkPublicMessage }
  }
}
```

Figure 12: AS Tier SELF_PROTECTING Policy

3.2.2 AS Tier Specification for Self-Protection

To protect the AS from insecure public messages we specified a self-management policy that handles the verification of any incoming public message. Thus, at this tier we specify a `SELF_PROTECTING` policy (one of the four self-CHOP policies [13]) to ensure protection from *insecure* public messages. Here we specified a single fluent mapped to an action via a mapping clause (see Figure 12).

The `inSecurityCheck` fluent is initiated by a `pub-licMessageIsComing` event and is terminated by one of the events, such as `publicMessageSecure` or `public-MessageInsecure`. Here the `inSecurityCheck` fluent is activated when an instance of the ASIP-specified `publicMessage` is sent to a recipient in the AS. Recall that any public message to be sent to a system recipient (e.g., a DMARF node) must be an instance of the ASSL `publicMessage` message (see Section 3.2.1). Therefore, in our specification model, the `publicMessageSecure` event will be activated anytime when a `publicMessage` is about to be received by the AS (by a recipient in that AS).

Figure 13 presents the specification of all the three events used to initiate and terminate the `inSecurity-Check` fluent. As it is depicted, both `publicMessageIn- secure` and `publicMessageSecure` are prompted when `thereIsInsecurePublicMessage`'s value has changed. Special `GUARDS` are specified to prevent those events be prompted when that metric is *valid* or *not valid* respectively [25]. The corresponding metric `thereIsIn-securePublicMessage` accepts only Boolean values and is valid when it holds `FALSE`. The same metric is set to `TRUE` or `FALSE` by the `checkPublicMessage` action. Here the metric is set to `TRUE` anytime when a new public insecure message has been discovered (see Appendix B).

The `checkPublicMessage` action is mapped to the `inSecurityCheck` fluent (see Figure 12). Here this action is performed anytime when the AS enters in a *security check* state (determined by the `inSecurityCheck` fluent). This action is intended to check how secure is the incoming

```
EVENTS { //these events are used in the fluents specification
  EVENT publicMessageIsComing {
    ACTIVATION { SENT { ASIP.MESSAGES.publicMessage } }
  }
  EVENT publicMessageInsecure {
    GUARDS { NOT METRICS.thereIsInsecurePublicMessage }
    ACTIVATION {
      CHANGED { METRICS.thereIsInsecurePublicMessage } }
  }
  EVENT publicMessageSecure {
    GUARDS { METRICS.thereIsInsecurePublicMessage }
    ACTIVATION {
      CHANGED { METRICS.thereIsInsecurePublicMessage } }
  }
} // EVENTS
```

Figure 13: AS Tier Events

```
senderIdentified = false;
FOREACH member in AES {
  IF ( NOT senderIdentified ) THEN
    senderIdentified =
            call member.ACTIONS.checkSenderCertificate
            (ASIP.MESSAGES.publicMessage.senderSignature)
  END
};
IF NOT senderIdentified THEN
  // makes the metric invalid and thus, triggers the attached
  // event and blocks all the operations with public messages
  set METRICS.thereIsInsecurePublicMessage.VALUE = true
END
```

Figure 14: AS checkPublicMessage Action – Partial Specification

`publicMessage`, which triggers the self-protecting policy by prompting the `publicMessageIs-Coming` event (see Figure 13). To do that, the `checkPub-licMessage` action calls for each AE in the AS a `check-SenderCertificate` action that must be specified in each AE (DMARF stage) (see Figure 14).

The `checkSenderCertificate` action returns `TRUE` if the `publicMessage` carries a valid digital signature (see Appendix B), i.e., the message is sent by a trusted sender (DMARF node). As depicted by Figure 14, if one of the AE returns `TRUE`, then the `publicMessage` is considered secure; otherwise, it is considered insecure. If the message is insecure the `thereIsInsecurePublic-Message` metric is set to `TRUE`, which blocks the IO operations over this message (see Section 3.2.1 and Appendix B).

3.2.3 AE Tier Specification for Self-Protection

At this tier we specify the self-protecting mechanism for private messages. Thus, for each AE (DMARF stage) we specify a `SELF_PROTECTING` self-management policy identical to the same policy specified at the AS tier (see Section 3.2.2). Note that this policy deals with private messages specified at the AEIP tier (the AE's private interaction protocol – see Appendix B).

Therefore, similarly to the same policy specified for the AS tier, the AE-level `SELF_PROTECTING` policy is specified with a single `inSecurityCheck` fluent mapped to a `checkPrivateMessage` action. The `inSecurity-Check` fluent is initiated by the `privateMessageIsCo-mming` event and terminated by the `privateMessageIs-Secure` event or by the `privateMessageSecure`

```
MANAGED_ELEMENTS {
  MANAGED_ELEMENT STAGE_ME {
    // checks if a node certificate is valid
    INTERFACE_FUNCTION checkNodeCertificate {
      PARAMETERS { ProxyCertificate theCertificate }
      RETURNS { Boolean }
    }
  }
}
```

Figure 15: AE STAGE_ME Managed Element

event. These events are similar to their homologous events specified at the AS tier (see Section 3.2.2), but dealing with the AEIP-specified `privateMessage` message and with the `thereIsInsecurePrivateMessage` metric at the AE-level (see Appendix B).

To perform the security checks of incoming private messages, the `checkPrivateMessage` action invokes the `checkSenderCertificate` action (recall that the same action is called by the `checkPublicMessage` action to check public messages – see Section 3.2.2). Internally, the `checkSenderCertificate` action calls a *managed element interface function* specified at the AEIP protocol to check proxy certificates (see Figure 15).

Recall (see Section 2.1) that managed elements provide special interface functions to control the DMARF system. Hence, as depicted by Figure 15, we expect the DMARF stage to verify whether a specific proxy certificate is valid and to return `TRUE` or `FALSE`. DMARF does that through the Java Data Security Framework (JDSF) [9, 11].

3.3 ASSL Self-Optimization Model for DMARF

The two major functional requirements applicable to large DMARF installations related to self-optimization are outlined below:

Training set classification data replication. A DMARF-based system may do a lot of multimedia data processing and number crunching throughout the pipeline. The bulk of I/O-bound data processing falls on the sample loading stage and the classification stage. The preprocessing, feature extraction, and classification stages also do a lot of CPU-bound number crunching, matrix operations, and other potentially heavy computations. The stand-alone local MARF instance employs dynamic programming to cache intermediate results, usually as feature vectors, inverse co-variance matrices, and other array-like data. A lot of this data is absorbed by the classification stages. In the case of the DMARF, such data may end up being stored on different hosts that run the classification service potentially causing re-computation of the already computed data on other classification host that did a similar evaluation already. Thus, the classification stage nodes need to communicate to exchange the data they have lazily acquired among all the classification members. Such data mirroring/replication would optimize a lot of computational effort on the end nodes.

Dynamic communication protocol selection. Additional aspect of self-optimization is automatic selection of the available most efficient communication protocol. E.g., if DMARF initially uses WebServices XML-RPC and later discovers all of its nodes can also communicate using say Java RMI, they can switch to that as their default protocol in order to avoid marshaling and de-marshaling heavy SOAP XML messages that are always a subject of a big overhead

```
SELF_OPTIMIZING {
  // DMARF enters in the Classification Stage
  FLUENT inClassificationStage {
    INITIATED_BY { EVENTS.enteringClassificationStage }
    TERMINATED_BY { EVENTS.optimizationSucceeded,
                    EVENTS.optimizationNotSucceeded }
  }
  MAPPING {
    CONDITIONS { inClassificationStage }
    DO_ACTIONS { ACTIONS.runGlobalOptimization }
  }
}
```

Figure 16: AS Tier SELF_OPTIMIZING Policy

even in the compressed form.

Here, the DMARF Classification stage is augmented with a self-optimizing autonomic policy. We used ASSL to specify this policy and generate implementation for the same. Appendix C presents a partial specification of the ASSL self-optimization model for ADMARF. As specified, the autonomic behavior is encoded in a special ASSL construct denoted as `SELF_OPTIMIZING` policy. The latter is specified at two levels - the global AS-tier level and the level of single AE (the AE-tier). The algorithm behind is described by the following elements:

- Any time when ADMARF enters in the Classification stage, a self-optimization behavior takes place.
- The Classification stage itself forces the stage nodes synchronize their latest cached results. Here each node is asked to get the results of the other nodes.
- Before proceeding with the problem computation, each stage node strives to adapt to the most efficient currently available communication protocol.

The following sections describe the ASSL specification of the self-optimization algorithm revealed here.

3.3.1 AS Tier Specification for Self-Optimization

At this tier we specify a system-level `SELF_OPTIMIZING` policy and the actions and events supporting that policy. As was mentioned, ASSL supports policy specification with special constructs called *fluents* and *mappings* [25]. Whereas the former are special states with conditional duration, the latter map actions to be executed when the system enters in such a state.

Figure 16 depicts the AS-tier specification of the `SELF_OPTIMIZING` policy. As we see the policy is triggered when the special fluent `inClassificationStage` is initiated. Here when ADMARF enters the Classification stage in its pipeline, an AS-level `enteringClassificationStage` event is prompted to initiate the corresponding `inClassificationStage` fluent.

Further, this fluent is mapped to an AS-level `run-GlobalOptimization` action (see Appendix C). This action iterates over all the Classification stage nodes specified as distinct AEs (see Section 3.3.2) and calls for each node a special AE-level `synchronizeResults` action (see Appendix C). In case of exception, the `optimizati-onNotSucceeded` event is issued; else the

`optimizati-onSucceeded` event is issued. Both events terminate the `inClassificationStage` fluent, and consecutively ADMARF exits the `SELF_OPTIMIZING` policy.

To distinguish the AEs from the other AEs in ADMARF, we specified the architecture topology of the system. For this we used the `ASARCHITECTURE` ASSL construct [25]. Appendix C presents the specification of the ADMARF architecture topology. Note that this is a partial specification depicting only two AEs. The full `ASARCHITECTURE` specification includes all the AEs of ADMARF. As depicted, we specified a special group of AEs called `CLASSF_STAGE` with members all the AEs representing the Classification stage nodes. This group allows the `runGlobalOptimization` action iterates over the stage nodes.

3.3.2 AE Tier Specification for Self-Optimization

At this tier we specified the `SELF_OPTIMIZING` policy for the Classification stage nodes. Here we specified for every node a distinct AE. (see Appendix C) presents the partial specification of two AEs, each representing a single node of the Specification stage. At this level, self-optimization concentrates on adapting the single nodes to the most efficient communication protocol. Similar to the AS-level policy specification (see Section 3.3.1), an `inCPAdaptation` fluent is specified to trigger such adaptation when ADMARF enters in the Specification stage. This fluent is initiated by the AS-level `enteringClassificationStage` event.

The same fluent is mapped to an `adaptCP` action to perform the needed adaptation. This action is specified as `IMPL`, i.e., requiring further implementation [25]. In ASSL, we specify `IMPL` actions to hide complexity via abstraction. Here, the `adaptCP` action is a complex structure, which explanation is beyond the scope of this paper. Therefore, we abstracted the specification of this action (through `IMPL`) and provided only the prerequisite *guard* conditions and prompted events.

4 Conclusion

In this article, we have presented ASSL specification models for autonomic features of ADMARF. To develop these features, we devised algorithms with ASSL for the pipelined stages of the DMARF's pattern recognition pipeline. The autonomic features were specified as special self-managing policies for self-healing, self-protecting, and self-optimizing in ADMARF. The ADMARF system (upon completion of the open-source implementation) will be able to fully function in autonomous environments, be those on the Internet, large multimedia processing farms, robotic spacecraft that do their own analysis, or simply even pattern-recognition research groups that can rely more on the availability of their systems that run for multiple days, unattended. Although not a fully complete specification model for ADMARF, we have attempted to provide didactic evidence of how ASSL can help us achieve desired automicity in DMARF.

Future work is concerned with further ADMARF development by including new autonomic features. For example, together with the full implementation and testing of the presented specification models, we intend to develop autonomic features covering the self-configuration aspects of ADMARF. These will help to construct an intelligent ADMARF system able to react automatically to dynamic requirements by finding possible solutions and applying those with no human interaction.

References

[1] DNSSEC.NET. DNSSEC: DNS Security Extensions Securing the Domain Name System, 2002–2011. `http://www.dnssec.net/`, last viewed November 2010.

[2] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[3] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM T. J. Watson Laboratory, Oct. 2001.

[4] IBM Corporation. An architectural blueprint for autonomic computing. Technical report, IBM Corporation, 2006.

[5] S. A. Mokhov. Choosing best algorithm combinations for speech processing tasks in machine learning using MARF. In S. Bergler, editor, *Proceedings of the 21st Canadian AI'08*, LNAI 5032, pages 216–221, Berlin Heidelberg, May 2008. Springer-Verlag.

[6] S. A. Mokhov. Towards security hardening of scientific distributed demand-driven and pipelined computing systems. In *Proceedings of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, pages 375–382. IEEE Computer Society, July 2008.

[7] S. A. Mokhov. Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 1288–1294, Turku, Finland, July 2008. IEEE Computer Society.

[8] S. A. Mokhov and M. Debbabi. File type analysis using signal processing techniques and machine learning vs. `file` unix utility for forensic analysis. In O. Goebel, S. Frings, D. Guenther, J. Nedon, and D. Schadt, editors, *Proceedings of the IT Incident Management and IT Forensics (IMF'08)*, LNI140, pages 73–85. GI, Sept. 2008.

[9] S. A. Mokhov, L. W. Huynh, and L. Wang. The integrity framework within the Java Data Security Framework (JDSF): Design refinement and implementation. In T. Sobh, K. Elleithy, and A. Mahmood, editors, *Novel Algorithms and Techniques in Telecommunications and Networking, Proceedings of CISSE'08*, pages 449–455. Springer, Dec. 2008. Printed in January 2010.

[10] S. A. Mokhov and R. Jayakumar. Distributed Modular Audio Recognition Framework (DMARF) and its applications over web services. In T. Sobh, K. Elleithy, and A. Mahmood, editors, *Proceedings of TeNe'08*, pages 417–422, University of Bridgeport, CT, USA, Dec. 2008. Springer. Printed in January 2010.

[11] S. A. Mokhov, F. Rassai, L. W. Huynh, and L. Wang. The authentication framework within the Java data security framework (JDSF): Design refinement and implementation. In T. Sobh, K. Elleithy, and A. Mahmood, editors, *Novel Algorithms and Techniques in Telecommunications and Networking, Proceedings of CISSE'08*, pages 423–429. Springer, Dec. 2008. Printed in January 2010.

[12] S. A. Mokhov and E. Vassev. Autonomic specification of self-protection for Distributed MARF with ASSL. In *Proceedings of C3S2E'09*, pages 175–183, New York, NY, USA, May 2009. ACM.

[13] R. Murch. *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall, 2004.

[14] Sun Microsystems, Inc. Java IDL. Sun Microsystems, Inc., 2004.

[15] Sun Microsystems, Inc. The Java web services tutorial (for Java Web Services Developer's Pack, v2.0). [online], Feb. 2006.

[16] The Planetary Society. Space topics: Voyager - the story of the mission, 2009. `http://planetary.org/explore/topics/spacemissions/voyager/objectives.html`.

[17] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. NASA's swarm missions: The challenge of building autonomous software. *IT Professional*, 6(5):47–52, 2004.

[18] E. Vassev. *ASSL: Autonomic System Specification Language – A Framework for Specification and Code Generation of Autonomic Systems*. LAP Lambert Academic Publishing, Nov. 2009. ISBN: 3-838-31383-6.

[19] E. Vassev and M. Hinchey. ASSL specification and code generation of self-healing behavior for NASA swarm-based systems. In *Proceedings of the 6th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'09)*, pages 77–86. IEEE Computer Society, 2009.

[20] E. Vassev and M. Hinchey. Modeling the image-processing behavior of the NASA Voyager mission with ASSL. In *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09)*, pages 246–253. IEEE Computer Society, 2009.

[21] E. Vassev, M. Hinchey, and J. Paquet. A self-scheduling model for NASA swarm-based exploration missions using ASSL. In *Proceedings of the Fifth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'08)*, pages 54–64. IEEE Computer Society, 2008.

[22] E. Vassev, M. G. Hinchey, and J. Paquet. Towards an ASSL specification model for NASA swarm-based exploration missions. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track*, pages 1652–1657. ACM, 2008.

[23] E. Vassev and S. A. Mokhov. Self-optimization property in autonomic specification of Distributed MARF with ASSL. In B. Shishkov, J. Cordeiro, and A. Ranchordas, editors, *Proceedings of ICSOFT'09*, volume 1, pages 331–335, Sofia, Bulgaria, July 2009. INSTICC Press.

[24] E. Vassev and S. A. Mokhov. Towards autonomic specification of Distributed MARF with ASSL: Self-healing. In *Proceedings of SERA 2010 (selected papers)*, volume 296 of *SCI*, pages 1–15. Springer, 2010.

[25] E. I. Vassev. *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[26] A. Wollrath and J. Waldo. Java RMI tutorial. Sun Microsystems, Inc., 1995–2005.

A ASSL Specification of DMARF Self-Healing

What follows is the complete initial ASSL specification of the self-healing aspect for the DMARF's pipeline stages [24].

```
// ASSL self-healing specification model for DMARF

AS DMARF {

  TYPES {  DMARFNode }

  ASSLO {
    SLO performance {
      FOREACH member in AES {
        member.AESLO.performance
      }
    }
  }

  ASSELF_MANAGEMENT {
    SELF_HEALING {
      // a performance problem has been detected
      FLUENT inLowPerformance {
        INITIATED_BY { EVENTS.lowPerformanceDetected }
        TERMINATED_BY {  EVENTS.performanceNormalized, EVENTS.performanceNormFailed }
      }
      MAPPING {
        CONDITIONS { inLowPerformance }
        DO_ACTIONS { ACTIONS.startSelfHealing }
      }
    }
  } // ASSELF_MANAGEMENT

  ACTIONS {
    ACTION IMPL startSelfHealing {
      GUARDS { ASSELF_MANAGEMENT.SELF_HEALING.inLowPerformance }
      TRIGGERS {
        IF NOT AES.STAGE_AE.AESLO.performance THEN
           AES.STAGE_AE.EVENTS.mustDoSelfHealing
        END
      }
```

```
    }
   } // ACTIONS

  EVENTS { // these events are used in the fluents specification
    EVENT lowPerformanceDetected { ACTIVATION { DEGRADED { ASSLO.performance} } }
    EVENT performanceNormalized { ACTIVATION { NORMALIZED { ASSLO.performance} } }
    EVENT performanceNormFailed { ACTIVATION { OCCURRED { AES.STAGE_AE.EVENTS.selfHealingFailed } } }
  } // EVENTS

} // AS DMARF

AES {

  AE STAGE_AE {

    VARS { DMARFNode nodeToRecover }

    AESLO {
      SLO performance {
        METRICS.numberOfFailedNodes
        AND
        METRICS.numberOfProblematicNodes
      }
    }

    AESELF_MANAGEMENT {
      SELF_HEALING {
        FLUENT inActiveSelfHealing {
          INITIATED_BY { EVENTS.mustDoSelfHealing }
          TERMINATED_BY { EVENTS.selfHealingSuccessful, EVENTS.selfHealingFailed }
        }
        FLUENT inFailedNodesDetected {
          INITIATED_BY { EVENTS.mustSwitchToNodeReplica }
          TERMINATED_BY { EVENTS.nodeReplicaStarted, EVENTS.nodeReplicaFailed }
        }
        FLUENT inProblematicNodesDetected {
          INITIATED_BY { EVENTS.mustFixNode }
          TERMINATED_BY { EVENTS.nodeFixed, EVENTS.nodeCannotBeFixed }
        }
        MAPPING {
          CONDITIONS { inActiveSelfHealing }
          DO_ACTIONS { ACTIONS.analyzeProblem }
        }
        MAPPING {
          CONDITIONS { inFailedNodesDetected }
          DO_ACTIONS { ACTIONS.startReplicaNode }
        }
        MAPPING {
          CONDITIONS { inProblematicNodesDetected }
          DO_ACTIONS { ACTIONS.fixProblematicNode }
        }
      }
    } // AESELF_MANAGEMENT

    AEIP {
      MANAGED_ELEMENTS {
        MANAGED_ELEMENT STAGE_ME {
          INTERFACE_FUNCTION countFailedNodes {
            RETURNS { Integer  }
          }
          INTERFACE_FUNCTION countProblematicNodes {
            RETURNS { Integer  }
          }
          // returns the next failed node
          INTERFACE_FUNCTION getFailedNode {
            RETURNS { DMARFNode  }
          }
          // returns the next problematic node
          INTERFACE_FUNCTION getProblematicNode {
            RETURNS { DMARFNode  }
          }
          // runs the replica of a failed nodee
```

```
      INTERFACE_FUNCTION runNodeReplica {
        PARAMETERS { DMARFNode  node }
        ONERR_TRIGGERS { EVENTS.nodeReplicaFailed }
      }
      // recovers a problematic node
      INTERFACE_FUNCTION recoverNode {
        PARAMETERS { DMARFNode  node }
        ONERR_TRIGGERS { EVENTS.nodeCannotBeFixed }
      }
    }
  }
} // AEIP

ACTIONS {
  ACTION analyzeProblem {
    GUARDS { AESELF_MANAGEMENT.SELF_HEALING.inActiveSelfHealing }
    VARS { BOOLEAN failed }
    DOES {
      IF METRICS.numberOfFailedNodes THEN
        AES.STAGE_AE.nodeToRecover = call AEIP.MANAGED_ELEMENTS.STAGE_ME.getFailedNode;
        failed = TRUE
      END
      ELSE
        AES.STAGE_AE.nodeToRecover = call AEIP.MANAGED_ELEMENTS.STAGE_ME.getProblematicNode;
        failed = FALSE
      END
    }
    TRIGGERS {
      IF failed THEN EVENTS.mustSwitchToNodeReplica END
      ELSE EVENTS.mustFixNode END
    }
    ONERR_TRIGGERS { EVENTS.selfHealingFailed }
  }

  ACTION startReplicaNode {
    GUARDS { AESELF_MANAGEMENT.SELF_HEALING.inFailedNodesDetected }
    DOES {  call AEIP.MANAGED_ELEMENTS.STAGE_ME.runNodeReplica(AES.STAGE_AE.nodeToRecover) }
    TRIGGERS {   EVENTS.nodeReplicaStarted }
  }

  ACTION fixProblematicNode {
    GUARDS { AESELF_MANAGEMENT.SELF_HEALING.inProblematicNodesDetected }
    DOES {  call AEIP.MANAGED_ELEMENTS.STAGE_ME.recoverNode(AES.STAGE_AE.nodeToRecover) }
    TRIGGERS {   EVENTS.nodeFixed}
  }
} // ACTIONS

EVENTS {
  EVENT mustDoSelfHealing { }
  EVENT selfHealingSuccessful {
    ACTIVATION {
      OCCURRED {   EVENTS.nodeReplicaStarted }
      OR
      OCCURRED {   EVENTS.nodeFixed }
    }
  }
  EVENT selfHealingFailed {
    ACTIVATION {
      OCCURRED {   EVENTS.nodeReplicaFailed }
    }
  }
  EVENT mustSwitchToNodeReplica {
    ACTIVATION {
      OCCURRED {   EVENTS.nodeCannotBeFixed }
    }
  }
  EVENT nodeReplicaStarted { }
  EVENT nodeReplicaFailed {  }
  EVENT mustFixNode { }
  EVENT nodeFixed {  }
  EVENT nodeCannotBeFixed {  }
} // EVENTS
```

```
    METRICS {
      // increments when a failed node has been discovered
      METRIC numberOfFailedNodes {
        METRIC_TYPE { RESOURCE  }
        METRIC_SOURCE {   AEIP.MANAGED_ELEMENTS.STAGE_ME.countFailedNodes }
        DESCRIPTION {"counts failed nodes in the MARF stage"}
        VALUE { 0 }
        THRESHOLD_CLASS {  Integer [0] } // valid only when holding 0 value
      }
      // increments when a problematic node has been discovered
      METRIC numberOfProblematicNodes {
        METRIC_TYPE { RESOURCE  }
        METRIC_SOURCE {   AEIP.MANAGED_ELEMENTS.STAGE_ME.countProblematicNodes }
        DESCRIPTION {"counts nodes with problems in the MARF stage"}
        VALUE { 0 }
        THRESHOLD_CLASS {  Integer [0] } // valid only when holding 0 value
      }
    }
  }
}
```

B ASSL Code Specification for DMARF Self-Protection

What follows is the complete initial ASSL specification of the self-protection aspect for the DMARF's pipeline stages [12].

```
// ASSL self-protecting specification model for DMARF

AS DMARF {

  TYPES {  ProxyCertificate }

  ASSELF_MANAGEMENT {
    // if a private message is detected as being insecure then
    // ignore it - the AE cannot neither receive nor resend it
    SELF_PROTECTING {
      // a new incoming message has been detected
      FLUENT inSecurityCheck {
        INITIATED_BY { EVENTS.publicMessageIsComing }
        TERMINATED_BY {  EVENTS.publicMessageSecure,
            EVENTS.publicMessageInsecure }
      }
      MAPPING {
        CONDITIONS { inSecurityCheck }
        DO_ACTIONS { ACTIONS.checkPublicMessage }
      }
    }
  } // ASSELF_MANAGEMENT

  ACTIONS {
    ACTION checkPublicMessage {
      GUARDS { ASSELF_MANAGEMENT.SELF_PROTECTING.inSecurityCheck }
      VARS { Boolean senderIdentified }
      DOES {
        senderIdentified = false;
        FOREACH member in AES {
          IF ( NOT senderIdentified ) THEN
            senderIdentified =
              call member.ACTIONS.checkSenderCertificate
              (ASIP.MESSAGES.publicMessage.senderSignature)
          END
        };
        IF NOT senderIdentified THEN
// makes the metric invalid and thus, triggers the attached event
// and blocks all the operations with public messages
          set METRICS.thereIsInsecurePublicMessage.VALUE = true
        END
        ELSE
```

```
// makes the metric valid and thus, triggers the attached event
// and unblocks all the operations with public messages
          set METRICS.thereIsInsecurePublicMessage.VALUE = false
        END
      }
      ONERR_DOES {
        // if error then treat the message as insecure
        set METRICS.thereIsInsecurePublicMessage.VALUE = true
      }
    }
   } // ACTIONS

  EVENTS { // these events are used in the fluents specification
    EVENT publicMessageIsComing {
      ACTIVATION { SENT { ASIP.MESSAGES.publicMessage } }
    }
    EVENT publicMessageInsecure {
      GUARDS { NOT METRICS.thereIsInsecurePublicMessage }
      ACTIVATION { CHANGED { METRICS.thereIsInsecurePublicMessage } }
    }
    EVENT publicMessageSecure {
      GUARDS { METRICS.thereIsInsecurePublicMessage }
      ACTIVATION { CHANGED { METRICS.thereIsInsecurePublicMessage } }
    }

  } // EVENTS

  METRICS {
    // set to true when a new public insecure message
    // has been discovered
    METRIC thereIsInsecurePublicMessage {
      METRIC_TYPE { QUALITY  }
      DESCRIPTION {"detects an insecure message in the AE"}
      VALUE { false }
      // valid only when holding false value
      THRESHOLD_CLASS {  Boolean [false] }
    }
  }
} // AS DMARF

ASIP {
  MESSAGES {
    MESSAGE publicMessage {
      SENDER { ANY }
      RECEIVER { AES.STAGE_AE }
      MSG_TYPE {   TEXT }
      PARAMETERS {  ProxyCertificate senderSignature }
    }
  }

  CHANNELS {
    CHANNEL publicLink {
      ACCEPTS {  MESSAGES.publicMessage }
      ACCESS { SEQUENTIAL }
      DIRECTION { INOUT }
    }
  }

  FUNCTIONS {
    //receive public messages if the message is secure
    FUNCTION receivePublicMessages {
      DOES {
        IF ( AS.METRICS.thereIsInsecurePublicMessage ) THEN
          MESSAGES.publicMessage << CHANNELS.publicLink
        END
      }
    }
    //send public messages if the message is secure
    FUNCTION sendPublicMessages {
      DOES {
        IF ( AS.METRICS.thereIsInsecurePublicMessage ) THEN
          MESSAGES.publicMessage >> CHANNELS.publicLink
```

```
        END
      }
    }
  }
}

AES {

  AE STAGE_AE {

    AESELF_MANAGEMENT {
      // if a private message is detected as being insecure then
      // ignore it - the AE cannot neither receive nor resend it
      SELF_PROTECTING {
        FLUENT inSecurityCheck {
          INITIATED_BY { EVENTS.privateMessageIsComming }
          TERMINATED_BY {  EVENTS.privateMessageSecure,
                      EVENTS.privateMessageInsecure }
        }
        MAPPING {
          CONDITIONS { inSecurityCheck }
          DO_ACTIONS { ACTIONS.checkPrivateMessage }
        }
      }
    } // AESELF_MANAGEMENT

    AEIP {
      MESSAGES {
        MESSAGE privateMessage {
          SENDER { ANY }
          RECEIVER { AES.STAGE_AE }
          MSG_TYPE {   TEXT }
          PARAMETERS {  ProxyCertificate senderSignature}
        }
      }

      CHANNELS {
        CHANNEL privateLink {
          ACCEPTS {  AEIP.MESSAGES.privateMessage }
          ACCESS { SEQUENTIAL }
          DIRECTION { INOUT }
        }
      }

      FUNCTIONS {
        //receive private messages if the message is secure
        FUNCTION receivePrivateMessages {
          DOES {
            IF ( AES.STAGE_AE.METRICS.thereIsInsecurePrivateMessage ) THEN
              AEIP.MESSAGES.privateMessage << AEIP.CHANNELS.privateLink
            END
          }
        }
        //send private messages if the message is secure
        FUNCTION sendPrivateMessages {
          DOES {
            IF ( AES.STAGE_AE.METRICS.thereIsInsecurePrivateMessage ) THEN
              AEIP.MESSAGES.privateMessage >> AEIP.CHANNELS.privateLink
            END
          }
        }
      }

      MANAGED_ELEMENTS {
        MANAGED_ELEMENT STAGE_ME {
          // checks if a node certificate is valid
          INTERFACE_FUNCTION checkNodeCertificate {
            PARAMETERS { ProxyCertificate  theCertificate }
            RETURNS { Boolean  }
          }
        }
      }
```

```
    } // AEIP

    ACTIONS {
      ACTION checkSenderCertificate {
        PARAMETERS {  ProxyCertificate theCertificate }
        RETURNS { Boolean }
        VARS { Boolean found }
        DOES {
          found = call AEIP.MANAGED_ELEMENTS.STAGE_ME.checkNodeCertificate
            (theCertificate);
          return found
        }
      }

      ACTION checkPrivateMessage {
        GUARDS { AESELF_MANAGEMENT.SELF_PROTECTING.inSecurityCheck }
        VARS { Boolean senderIdentified }
        DOES {
          senderIdentified = call ACTIONS.checkSenderCertificate
            ( AEIP.MESSAGES.privateMessage.senderSignature );
          IF NOT senderIdentified THEN
            // makes the metric invalid and thus, triggers the attached event
            // and blocks all the operations with private messages
            set METRICS.thereIsInsecurePrivateMessage.VALUE = true
          END
          ELSE
            // makes the metric valid and thus, triggers the attached event
            // and unblocks all the operations with private messages
            set METRICS.thereIsInsecurePrivateMessage.VALUE = false
          END
        }
        ONERR_DOES {
          // if error then treat the message as insecure
          set METRICS.thereIsInsecurePrivateMessage.VALUE = true
        }
      }
    } // ACTIONS

    EVENTS {
      EVENT privateMessageIsComming {
        ACTIVATION { SENT { AEIP.MESSAGES.privateMessage } }
      }
      EVENT privateMessageInsecure {
        GUARDS { NOT METRICS.thereIsInsecurePrivateMessage }
        ACTIVATION { CHANGED { METRICS.thereIsInsecurePrivateMessage } }
      }
      EVENT privateMessageSecure {
        GUARDS { METRICS.thereIsInsecurePrivateMessage }
        ACTIVATION { CHANGED { METRICS.thereIsInsecurePrivateMessage } }
      }
    } // EVENTS

    METRICS {
      // set to true when an insecure private message is discovered
      METRIC thereIsInsecurePrivateMessage {
        METRIC_TYPE { QUALITY  }
        DESCRIPTION {"detects an insecure message in the AE"}
        VALUE { false }
        // valid only when holding false value
        THRESHOLD_CLASS {  Boolean [false] }
      }
    }
  }
}
```

C ASSL Code Specification for DMARF Self-Optimization

What follows is the complete initial ASSL specification of the self-optimization aspect for the DMARF's pipeline stages [23].

```
// ASSL self-optimization specification model for DMARF

AS DMARF {

  ASSELF_MANAGEMENT {
    // DMARF strives to optimize by synchronizing cached
    // results before starting with the Classification Stage
    SELF_OPTIMIZING {
      // DMARF enters in the Classification Stage
      FLUENT inClassificationStage  {
        INITIATED_BY { EVENTS.enteringClassificationStage }
        TERMINATED_BY {  EVENTS.optimizationSucceeded,
            EVENTS.optimizationNotSucceeded }
      }
      MAPPING {
        CONDITIONS { inClassificationStage }
        DO_ACTIONS { ACTIONS.runGlobalOptimization }
      }
    }
  } // ASSELF_MANAGEMENT

  ASARCHITECTURE {
    AELIST {AES.CLASSF_STAGE_NODE_1, AES.CLASSF_STAGE_NODE_2}
    DIRECT_DEPENDENCIES {
      DEPENDENCY AES.CLASSF_STAGE_NODE_1 { AES.CLASSF_STAGE_NODE_2 }
      DEPENDENCY AES.CLASSF_STAGE_NODE_2 { AES.CLASSF_STAGE_NODE_1 }
    }
    GROUPS {
      GROUP CLASSF_STAGE {
        MEMBERS { AES.CLASSF_STAGE_NODE_1, AES.CLASSF_STAGE_NODE_2 }
      }
    }
  }

  ACTIONS {
    ACTION runGlobalOptimization {
      GUARDS { ASSELF_MANAGEMENT.SELF_OPTIMIZING.inClassificationStage }
      DOES {
        FOREACH member IN ASARCHITECTURE.GROUPS.CLASSF_STAGE.MEMBERS {
          call IMPL member.ACTIONS.synchronizeResults
        }
      }
      TRIGGERS {
        EVENTS.optimizationSucceeded
      }
      ONERR_TRIGGERS {
        // if error then report unsuccessful optimization
        EVENTS.optimizationNotSucceeded
      }
    }
   } // ACTIONS

  EVENTS { // these events are used in the fluents specification
    EVENT enteringClassificationStage { }
    EVENT optimizationSucceeded { }
    EVENT optimizationNotSucceeded { }
  } // EVENTS

} // AS DMARF

AES {

  AE CLASSF_STAGE_NODE_1 {

    AESELF_MANAGEMENT {
      SELF_OPTIMIZING {
        FLUENT inCPAdaptation {
          INITIATED_BY { AS.EVENTS.enteringClassificationStage }
          TERMINATED_BY {  EVENTS.cpAdaptationSucceeded,
              EVENTS.cpAdaptationNotSucceeded }
        }
        MAPPING {
```

```
          CONDITIONS { inCPAdaptation }
          DO_ACTIONS { ACTIONS.adaptCP }
        }
      }
    }

    ACTIONS {
      ACTION IMPL synchronizeResults {
        GUARDS { AS.ASSELF_MANAGEMENT.SELF_OPTIMIZING.
            inClassificationStage
        }
      }
      ACTION IMPL adaptCP {
        GUARDS { AESELF_MANAGEMENT.SELF_OPTIMIZING.inCPAdaptation }
        TRIGGERS { EVENTS.cpAdaptationSucceeded }
        ONERR_TRIGGERS { EVENTS.cpAdaptationNotSucceeded }
      }
    } // ACTIONS

    EVENTS { // these events are used in the fluents specification
      EVENT cpAdaptationSucceeded { }
      EVENT cpAdaptationNotSucceeded { }
    } // EVENTS
  }

  AE CLASSF_STAGE_NODE_2 {

    AESELF_MANAGEMENT {
      SELF_OPTIMIZING {
        FLUENT inCPAdaptation {
          INITIATED_BY { AS.EVENTS.enteringClassificationStage }
          TERMINATED_BY {  EVENTS.cpAdaptationSucceeded,
              EVENTS.cpAdaptationNotSucceeded }
        }
        MAPPING {
          CONDITIONS { inCPAdaptation }
          DO_ACTIONS { ACTIONS.adaptCP }
        }
      }
    }

    ACTIONS {
      ACTION IMPL synchronizeResults {
        GUARDS { AS.ASSELF_MANAGEMENT.SELF_OPTIMIZING.
            inClassificationStage
        }
      }
      ACTION IMPL adaptCP {
        GUARDS { AESELF_MANAGEMENT.SELF_OPTIMIZING.inCPAdaptation }
        TRIGGERS { EVENTS.cpAdaptationSucceeded }
        ONERR_TRIGGERS { EVENTS.cpAdaptationNotSucceeded }
      }
    } // ACTIONS

    EVENTS { // these events are used in the fluents specification
      EVENT cpAdaptationSucceeded { }
      EVENT cpAdaptationNotSucceeded { }
    } // EVENTS}
  }
}
```